공학석사 학위논문

머신 러닝 어플리케이션을 위한 스냅샷 기반의 연산 오프로딩

2017년 8월

서울대학교 대학원

전기정보공학부

정인창

머신 러닝 어플리케이션을 위한 스냅샷 기반의 연산 오프로딩

지도교수 문 수 묵

이 논문을 공학석사 학위논문으로 제출함
2017년 8월

서울대학교 대학원
전기정보공학부
정인창

정인창의 석사 학위논문을 인준함
2017년 8월

위 원 장 백 윤 흥 (인)

부위원장 문 수 묵 (인)

위 원 이 혁 재 (인)

국문초록

머신러닝 기술은 데이터를 학습하고 문제의 답을 추론하기 위해 복잡한 연산과 방대한 데이터를 요구한다. 이런 머신러닝 기술을 저사양 임베디드 기기에서 활용하기 위해 오프로딩 기반 머신러닝이 제안되었다. 연산 오프로딩이란 임베디드 기기에서 복잡한 연산을 동적으로 서버를 통해 수행하는 방식이다. 본 논문에서는 웹 어플리케이션을 대상으로 스냅샷 기반 연산 오프로딩을 사용하였다. 스냅샷이란 수행 중인 웹 어플리케이션의 상태를 또 다른 웹 어플리케이션의 형태로 저장하고 복원하는 기술이다. 스냅샷 기반 연산 오프로딩을 머신러닝 웹 어플리케이션에 적용 시 두 가지 이슈가 발생한다. 하나는 웹에서 이미지를 처리하는 캔버스 객체의 전송 문제이며 다른 하나는 크기가 큰 머신러닝 모델 전송 문제이다. 본 연구에서는 두 가지 이슈를 해결하여 스냅샷 기반 오프로딩을 통한 머신 러닝 웹 어플리케이션의 올바른 동작과 성능 향상을 확인하였다. 두 가지 이슈를 해결하여 실제 머신러닝 웹 어플리케이션에서 추론 시간을 측정하였다. 측정 결과 클라이언트 수행 시간 대비 오프로딩 시 3-3.5배 성능 향상을 확인하였다.

주요어 : 연산 오프로딩, 스냅샷, 머신 러닝, 웹 어플리케이션, 캔버스
학 번 : 2015-22809

목 차

표 목 차

그 림 목 차

제 1 장 서 론

모바일 기기의 하드웨어 성능이 발전하면서 모바일 기기에서도 다소 복잡한 어플리케이션 수행이 가능해지고 있다. 그러나 최근 여러 다양한 분야에서 사용되고 있는 머신러닝 기술은 매우 많은 연산을 요구하기 때문에 최신의 모바일 기기 환경에서도 수행 시간이 매우 길어지는 문제가 발생한다. 이를 해결하기 위해 모바일 기기의 수행을 돕는 여러 가지 연구가 진행되었으며 그 중 한 방법으로 연산 오프로딩 기술이 제안된 바 있다.

연산 오프로딩이란 하드웨어 성능이 부족한 기기에서 수행해야 할 복잡한 연산을 서버로 보내 대신 수행하게 한 뒤 결과를 받아서 반영하는 기술이다. 연산 오프로딩을 위해서는 동작중인 어플리케이션을 모바일 기기에서 서버로 이주시키고 연산을 수행한 뒤 다시 모바일 기기로 이주시켜야 한다. 이 때 어플리케이션의 상태 정보를 전송하는 과정이 까다로워 기존 방식에서는 프로그래밍이 어렵고 비효율적인 접근 방법들이 사용되었다[3][4].

최근 이러한 문제를 해결하기 위해 웹 기술의 이식성을 활용해 웹 어플리케이션의 상태 정보를 서버-클라이언트 간에 간단히 전송할 수 있는 스냅샷 기반 연산 오프로딩이 제안되었다[2]. 스냅샷 기반 연산 오프로딩은 웹 어플리케이션의 상태를 HTML, CSS, JavaScript로 이루어진 또 다른 웹 어플리케이션의 형태로 저장하는 스냅샷 기술을 이용하고 있다. 스냅샷은 단순히 수행되는 것 만으로도 웹 어플리케이션의 상태를 복원하고 수행을 계속 할 수 있어 복잡한 어플리케이션의 상태 정보 전송 과정이 단순화 되었고, 따라서 기존 방식에 비해 프로그래밍이 쉽고 효율적으로 작성 될 수 있다[1].

하지만 기존의 스냅샷은 웹 어플리케이션의 일부 상태를 제대로 저장하지 못하는 문제가 있다. 스냅샷은 웹 어플리케이션의 화면을 DOM

tree를 저장하여 복구한다. 대부분의 DOM 객체가 화면 요소의 모든 정보를 가지고 있기 때문에 DOM tree만 저장해도 화면을 완벽하게 복구할 수 있다. 하지만 JavaScript를 이용하여 화면에 그림을 그리는 요소인 HTML5 캔버스의 경우 화면을 구성하는데 필요한 정보를 DOM 객체가 아닌 브라우저 엔진 내부에서 별도로 관리한다. 즉 캔버스의 DOM 객체만 저장해서는 캔버스를 복구 할 수 없기 때문에, 기존의 스냅샷은 캔버스를 사용하는 웹 어플리케이션의 화면을 완벽하게 복구 할 수 없다. 캔버스가 화면을 쉽게 조작할 수 있다는 장점으로 이미지에 대한 머신러닝 웹 어플리케이션에 많이 사용되고 있어, 머신러닝 웹 어플리케이션에는 기존의 스냅샷 연산 오프로딩을 적용할 수 없다. 또한 스냅샷은 웹 어플리케이션의 상태를 코드 형태로 저장하다 보니 머신러닝의 모델과 같이 정보의 크기가 큰 데이터들을 비효율적으로 저장되는 문제도 있었다. 이러한 문제들로 머신러닝 웹 어플리케이션에 기존의 스냅샷 기반 연산 오프로딩을 적용할 수 없는 문제가 발생하였다.

본 논문에서는 기존의 스냅샷 기반 연산 오프로딩이 머신러닝 웹 어플리케이션에서 적용하면 발생하는 두 가지 문제를 해결하여 머신러닝 웹 어플리케이션에서도 스냅샷 기반 연산 오프로딩을 적용할 수 있도록 하였다.

본 논문의 구성은 다음과 같다. 1장의 서론에 이어 2장에서는 스냅샷 기반의 연산 오프로딩 방법 및 문제점을 설명한다. 3장에서는 스냅샷에 캔버스 정보를 저장 및 복구하는 방법을 제시한다. 4장에서는 스냅샷 기반 연산 오프로딩에서 머신러닝 모델을 전송 및 복구하는 방법을 제시할 것이다. 5장에서는 캔버스와 머신러닝 기법을 사용하는 웹 어플리케이션을 대상으로 제안한 기술을 사용하여 결과를 살펴볼 것이다. 마지막 6장에서는 결론과 향후 방향에 대해 논의할 것이다.

제 2 장 스냅샷 기반 연산 오프로딩

제 1 절 스냅샷

스냅샷이란 웹 어플리케이션의 상태 정보를 HTML, CSS, JavaScript로 이루어진 또 다른 웹 어플리케이션의 형태로 저장하는 기술이다. 스냅샷에서 저장하는 웹 어플리케이션의 상태 정보는 크게 DOM tree 정보와 JavaScript 상태 정보로 나눌 수 있다. DOM(Document Object Model)이란 HTML 문서를 처리하는 인터페이스로 HTML 문서의 구조화된 표현양식을 트리형태로 제공한다. HTML 요소들은 파싱될 때 DOM 객체가 생성되며 생성된 DOM 객체는 트리 구조를 이루고, 브라우저는 DOM tree를 이용하여 화면을 표시한다. JavaScript 상태 정보는 JavaScript 객체, 이벤트 정보, execution context, scope chain 등 JavaScript의 언어적인 특성을 위한 다양한 정보를 가지고 있다. 스냅샷은 브라우저의 루트 객체인 window 객체에서 모든 하위 객체들을 탐색하면서 현재 JavaScript 상태와 DOM tree를 저장하게 된다. 추가적으로 브라우저 내부 이벤트 정보를 살펴서 어플리케이션에서 등록한 이벤트 정보를 저장한다[1]. 스냅샷의 가장 큰 장점은 웹 어플리케이션의 상태를 쉽게 저장하고 복구할 수 있다는 점이다. 이러한 장점을 이용하여 복잡한 웹 어플리케이션을 대상으로 스냅샷 기반 연산 오프로딩 기법이 등장하였다[2].

제 2 절 스냅샷 기반 연산 오프로딩

스냅샷 기반 연산 오프로딩을 사용하기 위해선 우선 복잡한 연산 이벤트를 오프로딩 이벤트로 등록해야 한다. 이를 위해 해당 프레임워크에

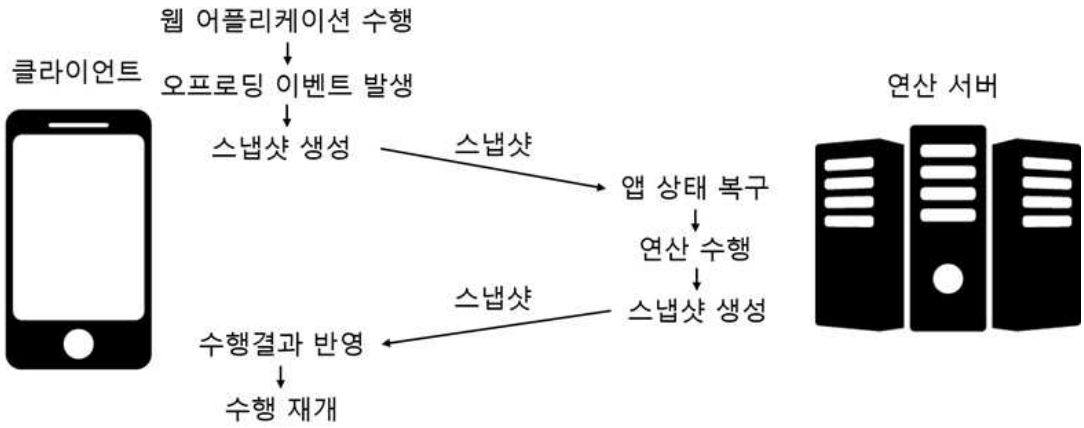

그림 1 스냅샷 기반 연산 오프로딩 과정

서는 간단한 JavaScript 함수를 제공한다[2]. 어플리케이션 제작자는 해당 함수를 이용해 복잡한 연산 이벤트를 오프로딩 이벤트로 등록 할 수 있다.

그림1은 스냅샷 기반 연산 오프로딩 과정을 나타낸 것이다. 웹 어플리케이션이 실행되면 클라이언트와 서버가 웹 소켓을 통해 연결된다. 그 후 등록된 오프로딩 이벤트가 발생하면 현재의 상태를 스냅샷으로 저장하고, 생성된 스냅샷을 서버로 전송 한 뒤 기다린다. 서버는 스냅샷을 받아 수행하여 어플리케이션의 상태를 복원 한 뒤 클라이언트가 요청한 복잡한 연산을 수행한다. 연산이 끝나면 연산 결과가 반영 된 스냅샷을 생성한 뒤 클라이언트로 보낸다. 클라이언트는 스냅샷을 수행하여 연산 결과를 반영한 뒤 어플리케이션의 수행을 계속해서 진행하게 된다.

그림2는 오프로딩을 사용하는 간단한 웹 어플리케이션과 해당 어플리케이션의 스냅샷 예제이다. 그림 2의 App.html은 버튼 요소의 클릭 이벤트에 오프로딩 할 복잡한 연산 이벤트 foo를 등록하는 어플리케이션 코드이다. Foo는 복잡한 연산을 하고 연산 결과를 새로운 DOM 객체 내부에 표시를 하여 사용자에게 결과를 알려준다. App.html을 실행시키고 버튼을 클릭하게 되면 오프로딩 이벤트가 발생하게 되고 현재의 상태를 스냅샷으로 저장하게 된다.

Snpashot-client-to-server.html은 클라이언트에서 생성된 스냅샷의

App.html

```
<html>
<head>
...
<script src="offloading_manager.js"></script>
</head>
<body>
<img id="img" src="faces.jpg"></img>
<button id="btn">Offload</button>
<script>
var foo = function(){
    var result;
    // computation intensive job to calculate result
    ...
    var new_div = document.createElement("div");
    new_div.innerHTML = result;
    document.body.appendChild(new_div);
}
var button = document.getElementById("btn");
addOffloadedEventListener(btn, "click", foo);
</script>
</body>
</html>
```

Snpashot-client-to-server.html

```
<html>
<head>
...
<script src="snapshot_manager.js"></script>
</head>
<body>
<img id="img" src="faces.jpg"></img>
<button id="btn">Offload</button>
<script>
(function(){
    // Restore Application states
    // (ex. objects, global variables, and event info)
    ...
    dispatchEvent();
})();
</script>
</body>
</html>
```

Snapshot-server-to-client.js

```
(function(){
    // Apply Computation Result
    var result =... // result of the computation;
    var new_div = document.createElement("div");
    new_div.innerHTML = result
    document.body.appendChild(new_div);
    ...
})();
```

그림 2 웹 어플리케이션과 스냅샷 예제 코드

예제이다. 코드를 살펴보면 우선 스냅샷과 관련된 프레임워크를 삽입한 뒤 DOM 요소들을 복구한다. 그 후 JavaScript의 objects, global variables, functions을 복구한 뒤 마지막으로 복잡한 연산 이벤트(foo)를 발생시켜 수행하게 된다. 서버는 해당 스냅샷을 클라이언트에서 받아서 수행하여 어플리케이션의 상태를 복구한 뒤 복잡한 연산 이벤트를 수행하게 된다. 연산 이벤트가 끝나면 서버는 수행결과가 반영된 스냅샷을 생성하게 된다.

Snapshot-server-to-client.js는 서버에서 연산을 한 뒤 생성된 스냅샷의 예제이다. 서버에서 생성 된 스냅샷은 수행 결과만 반영하면 되기 때문에 결과를 반영해주는 JavaScript 코드로만 이루어져 있다. 해당 코드를 살펴보면 foo와 비슷하게 DOM 객체를 만들어 결과를 표시해주는 코드가 존재한다. 클라이언트는 서버로부터 해당 스냅샷을 받아서 수행하여 연산 결과를 반영한다. 이처럼 스냅샷을 이용하면 서버와 클라이언트 간에 웹 어플리케이션의 상태를 쉽게 주고받을 수 있어, 큰 제약 없이 웹 어플리케이션에 연산 오프로딩 기술을 적용 할 수 있다.

하지만 기존의 스냅샷은 웹 어플리케이션의 일부 상태를 제대로 저장하지 못하는 문제가 있다. 앞서 설명했듯이 스냅샷은 현재 화면을 나타내기 위해 DOM tree를 저장한다. DOM tree를 저장하는 이유는 대부분의 DOM 객체가 화면 요소의 정보를 모두 가지고 있기 때문이다. 하지만 HTML5에서 새로 추가된 캔버스의 경우 DOM 객체가 캔버스의 모든 정보를 가지고 있지 않다. 캔버스는 JavaScript를 이용하여 웹 페이지나 웹 어플리케이션 상에 그림을 그리는 HTML 요소이다. 캔버스의 DOM 객체는 캔버스 자체의 크기나 위치, 스타일에 대한 정보만 가지고 있고 내부 이미지와 렌더링 정보들을 가지고 있지 않다. 그러므로 현재의 스냅샷을 이용하여 캔버스를 저장 후 복구하면 캔버스 요소 자체는 복구가 되지만 캔버스 내부 이미지나 그리는 방법들에 대해서는 제대로 복구가 되지 않는다. 캔버스가 화면을 쉽게 조작할 수 있어 이미지에 대한 머신러닝 웹 어플리케이션에 많이 사용되고 있는데, 이러한 머신러닝 웹 어플리케이션에는 기존의 스냅샷 연산 오프로딩을 적용할 수 없다.

또한 스냅샷 기반 연산 오프로딩을 사용하는 웹 어플리케이션은 기존의 서버-클라이언트 구조와는 다르게 클라이언트에서 학습된 모델을 가지고 있게 된다. 그 후 inference와 같은 연산이 복잡한 이벤트가 발생하면 모델과 데이터 등을 스냅샷 형태로 저장하여 서버에 보내게 된다. 일반적으로 머신러닝 어플리케이션에서 weight와 parameter와 같은 학습된 모델 정보의 크기는 어플리케이션의 다른 상태 정보에 비해 매우 크다. 스냅샷의 크기가 커질수록 오프로딩의 성능이 저하되기 때문에 모델 정보의 크기가 오프로딩 성능에 영향을 미치는 중요한 요소가 된다. 스냅샷은 앞서 설명했듯이 HTML, CSS, JavaScript 코드 형태로 이루어져 있다. 즉 스냅샷에 모델을 저장하게 되는 경우 모델 정보를 문자열 형태로 저장하게 된다. 이 경우 모델 정보를 바이너리 형태로 저장했을 때에 비해 크기가 수 배 증가하게 된다. 따라서 모델 정보를 스냅샷에 포함시키는 경우 오프로딩의 성능이 급격하게 저하되는 문제가 발생한다. 따라서 기존의 스냅샷 기반 연산 오프로딩 기법은 머신러닝 기술을 사용하는 웹 어플리케이션에는 적용하기 힘든 문제가 있다.

제 3 장 캔버스 저장 방법

캔버스는 HTML5에서 새로 추가된 요소로 JavaScript를 이용하여 웹 페이지나 어플리케이션에 그림을 그리는데 사용된다. 그림 3는 캔버스를 이용해 간단한 도형과 글자를 그린 예시이다. 캔버스 태그를 이용해 캔버스를 선언하면 그래픽을 그릴 수 있는 컨테이너가 만들어지게 되고 JavaScript를 이용하면 컨테이너에 여러 가지 그림을 그릴 수 있다. 간단하게는 도형 그리기, 글자 쓰기부터 그래프 그리기, 사진 합성, 애니메이션 제작 등을 할 수 있다. 별도의 라이브러리 없이 HTML과 JavaScript만으로 복잡한 그래픽을 쉽게 처리할 수 있어서 널리 사용되고 있다.

```
<html>
<head>
</head>
<body>
<canvas id="myCanvas" width="200" height="200"></canvas>
<script>
var canvas = document.getElementById("myCanvas");
var ctx = cavnas.getContext("2d");
ctx.fillStyle = "#60ff00";
ctx.fillRect(50,50,100,100);
ctx.font = "bold 20px Arial";
ctx.strokeText("CANVAS", 60, 100);
</script>
</body>
</html>
```

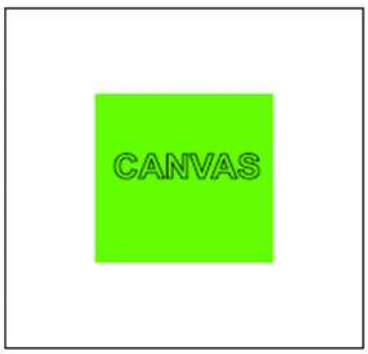

그림 3 캔버스 예시

앞서 설명했듯이 캔버스 DOM 객체는 캔버스의 크기와 위치에 관련된 정보만을 가지고 있다. 그림 3의 예시에서 DOM 객체를 저장하면 캔버스의 높이와 넓이, ID 그리고 화면에서 캔버스의 위치가 저장이 된다. DOM 객체만 저장하여 복구하는 경우 화면에 빈 캔버스가 생성이 된다. 내부 이미지까지 복구하기 위해선 캔버스의 다른 정보를 추가로 저장해 주어야 한다. 브라우저 내부에서 캔버스 객체는 ImageData와 Rendering

그림 4 그림 3의 캔버스 Data URL

Context 객체를 가지고 있다. ImageData는 현재 캔버스 내부 이미지에 대한 정보를 가지고 있으며 rendering context는 렌더링에 대한 정보를 가지고 있다[5]. 캔버스를 저장할 땐 DOM 객체뿐만 아니라 위 두 객체를 추가로 저장해야 캔버스를 온전히 복구 할 수 있다.

제 1 절 ImageData

ImageData 객체는 캔버스의 각 픽셀의 RGBA 값을 배열 형식으로 가지고 있다. 이를 코드 형태인 스냅샷과 같은 형식으로 저장하기 위해서는 우선 문자열 형태로 변환을 해주어야 한다. 배열 자체를 문자열로 변환해서 저장하는 방법도 있지만 이 경우 바이트 크기가 매우 커지기 때문에 인코딩하여 문자열로 변환하여야 한다. 본 논문에서는 캔버스 API인 toDataURL 함수를 이용하여 base64 인코딩[6] 방법을 사용하는 data url[7] 형식으로 저장한다. Data url 형식이란 data: 스킴이 접두어로 붙은 URL로 작은 파일을 문서 내에 인라인으로 삽입 할 수 있게 해준다. 또한 크기를 줄이기 위해 원본 데이터를 JPEG 형식으로 압축해서 저장한다. 그림 4은 그림 3의 캔버스 이미지를 data url 형식으로 변환한 예시이다.

캔버스의 이미지를 data url 형식으로 변환하여 스냅샷에 저장하면 현재 이미지를 쉽게 복구할 수 있다. 하지만 이미지 데이터를 인코딩하여 문자열로 변환하였기 때문에 이미지 데이터에 비해서 바이트 크기가 증

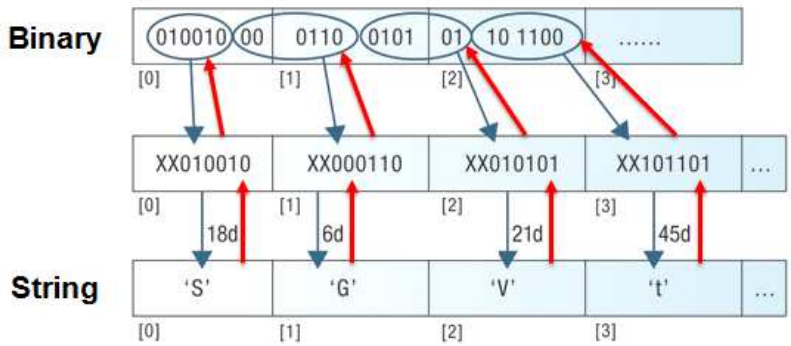

그림 5 Base64 인코딩 및 디코딩 방식

가하게 된다. 그림 5와 같이 Base64 인코딩 방식은 데이터를 6비트씩 나누어서 문자 코드에 영향을 받지 않는 공통 ASCII 영역의 문자열로 인코딩하는 방식이다. JavaScript는 ASCII 문자 하나당 8비트이기 때문에 base64 방식으로 인코딩을 하는 경우 이미지 데이터가 4/3배로 증가한다. 즉 이미지 데이터를 인코딩하여 스냅샷에 포함하는 경우 원본 데이터에 비해 바이트 크기가 증가한다. 보통 이미지 데이터가 스냅샷의 대부분을 차지하기 때문에 이미지 데이터 크기의 증가는 스냅샷 크기의 급격한 증가로 이어진다. 스냅샷의 크기가 커지는 경우 네트워크 시간이 증가해서 오프로딩의 성능이 저하되는 문제가 발생한다.

이 문제를 해결하고자 캔버스 이미지 데이터를 문자열 형식으로 스냅샷에 포함하지 않고, 바이너리 형식으로 스냅샷과 분리하여 별도로 전송하는 방법을 고안하였다. Base64로 인코딩 된 DataURL을 다시 디코딩하여 바이너리 형식으로 전송하였다. 대신 서버에서 어떤 캔버스의 이미지인지 쉽게 알 수 있도록 데이터 앞부분에 DOM tree상의 주소를 추가하였다. 캔버스 이미지 데이터를 스냅샷과 분리하는 경우, 캔버스 오브젝트의 숫자만큼 서버와 클라이언트 사이에 전송을 추가적으로 하게 되는 단점이 발생하지만 인코딩으로 인한 크기 증가가 발생하지 않기 때문에 총 전송 시간은 감소하여 전체 성능은 향상된다.

제 2 절 Rendering Context

Rendering Context는 캔버스에 그림을 그리는 함수들과 어떻게 그릴 것인지에 대한 렌더링 정보들을 가지고 있다. 그림 3의 예시에서 ctx 변수가 2d rendering context이며 fiilStyle과 font가 렌더링 정보에 해당한다. 해당 캔버스를 정확히 복구하기 위해서는 이미지 데이터뿐만 아니라 렌더링 정보들도 복구를 해야 한다. 렌더링 정보를 저장하지 않을 경우 서버와 클라이언트 사이에 렌더링 값이 달라지기 때문에 의도대로 그림이 그려지지 않을 수 있다. 표 1은 그림 3의 예시에서 rendering context가 가지고 있는 렌더링 정보의 일부이다. Rendering context는 캔버스의 getContext 함수를 이용하여 얻을 수 있다. 캔버스를 저장할 때 rendering context 객체를 함께 스냅샷에 저장하여 렌더링 정보가 제대로 복구 될 수 있도록 한다.

fillStyle	#60ff00	lineWidth	1
font	bold 20px	shadowBlur	0
globalAlpha	1	shadowColor	0,0,0,0
lineCap	butt	strokeStyle	#000000

표 1 그림 3의 렌더링 정보 예시

제 4 장 머신러닝 모델 전송 방법

현재 웹 어플리케이션에서 주로 사용되는 머신러닝 프레임워크를 살펴보면 브라우저에서 학습을 하지 않고 주로 caffe나 keras와 같은 프레임워크로 학습된 모델을 가져와 사용하는 형태가 많다. 즉 클라이언트에 바이너리 형태로 저장된 모델이 존재하고 이를 불러와서 사용하는 경우가 대다수이다. 따라서 스냅샷 생성 시 모델이 저장된 경로를 알 수 있으면 모델을 스냅샷에 포함하지 않고 바이너리 형태로 전송 할 수 있다. 이를 위해 연산 오프로딩 프레임워크에 아래와 같은 간단한 JavaScript 함수를 구현하였다.

addSkippedModel(obj, path, load_function)

해당 함수를 사용하여 모델 객체와 모델이 저장되어 있는 경로를 알려주면 스냅샷 생성 시 모델 객체를 만났을 때 모델을 저장하지 않고 경로에 저장되어 있는 바이너리 파일을 서버에 전송하도록 하였다. 또한 서버에서 모델을 다시 불러올 수 있도록 모델을 불러오는 함수를 받아서 스냅샷 생성 시 추가로 삽입하였다. 따라서 서버에서 스냅샷을 수행하면 모델을 다시 불러와서 마치 스냅샷에 모델 정보가 포함되어 있는 것처럼 사용할 수 있다. 위 함수를 이용하면 모델의 정보를 안전하게 서버로 보낼 수 있을 뿐만 아니라 스냅샷의 크기도 줄여 오프로딩의 성능을 증가시킬 수 있다.

만약 추론 어플리케이션과 같이 어플리케이션 수행 시 모델이 변하지 않는다면 모델을 어플리케이션 시작 시점에서 미리 서버에 전송하여 오프로딩 성능을 더 향상시킬 수 있다. 또한 모델을 미리 전송할 때 별도의 thread를 부여하여 모델을 전송하는 경우 어플리케이션의 수행을 막지 않고 모델을 전송할 수 있다. 웹 어플리케이션의 경우 Web Worker

를 통해 별도의 thread에 모델 전송 작업을 전달 할 수 있다. 모델을 전송하는 동안 추론 요청이 들어오는 경우 클라이언트에서 추론을 수행하도록 한다. 모델 전송이 완료된 후 들어오는 추론 요청에 대해서는 연산 오프로딩 기법을 적용하여 추론하도록 한다.

제 5 장 실험 및 결과

제 1 절 실험 환경

클라이언트는 ARM 임베디드 보드인 odroid xu4[8]에서 수행하였다. Odroid xu4는 2.0 GHz 코어 4개와 1.5GHz 코어 4개를 가지고 있으며 2GB 메모리를 가지고 있으며, 운영체제는 Ubuntu 14.04 환경에서 수행하였다. 서버는 3.4GHz 쿼드코어 CPU와 16GB메모리를 가지고 있으며 운영체제는 Ubuntu 12.04 환경의 데스크톱 PC에서 수행하였다. 네트워크 환경은 이더넷 환경에서 실험하였다.

실험 어플리케이션은 캔버스에 그려진 이미지를 추론하는 어플리케이션들로 실험을 하였으며, 실험에 사용된 모델은 GoogLeNet[9], AgeNet[10], GenderNet[10]을 사용하였다. GoogLeNet은 입력 이미지가 어떤 이미지인지 추론하는 모델이며, AgeNet은 입력 이미지의 사람의 나이를 추론하는 모델이며, GenderNet은 입력 이미지의 사람의 성별을 추론하는 모델이다. 사용한 프레임워크는 caffe.js[11]로 caffe로 학습된 모델을 브라우저로 가져와 사용할 수 있도록 하는 프레임워크이다. 실험은 크게 캔버스 저장 방법에 따른 효과를 측정하는 실험과 모델 전송 방법에 따른 효과를 측정하는 실험으로 나누어서 진행하였다. 각 실험마다 추론 수행시간을 측정하였으며 클라이언트와 서버에서 수행한 시간과 비교하였다.

제 2 절 캔버스 저장 방법에 따른 측정 결과

그림 6는 캔버스 저장 방법에 따른 추론 수행시간이다. 이 때 모델 전

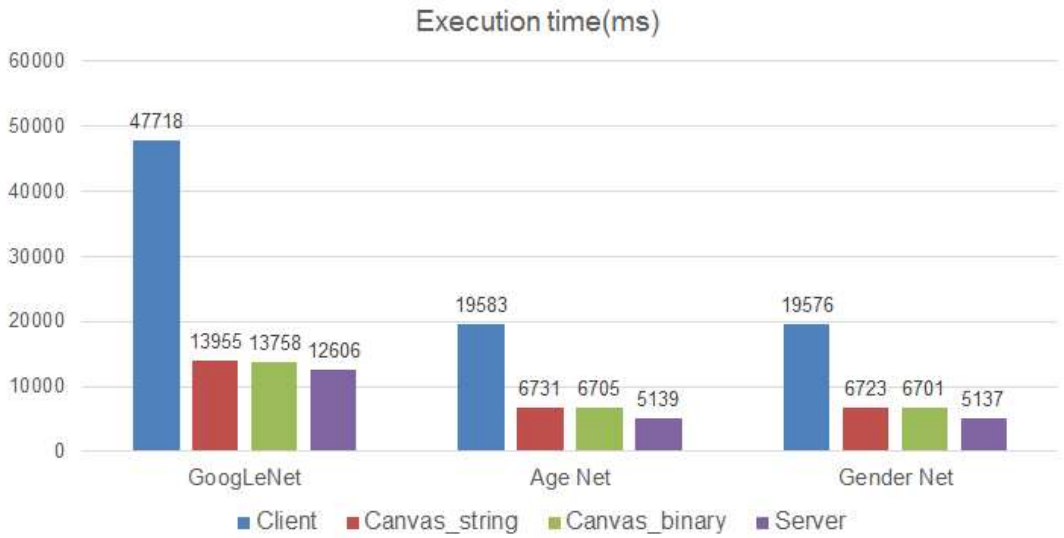

그림 6 캔버스 저장 방법에 따른 추론 수행시간

송 방법은 Web worker를 통해 미리 전송하는 방식으로 측정하였다. Canvas string save는 캔버스 이미지 데이터를 문자열 형태로 저장했을 때 수행시간이며, Canvas binary save는 캔버스 이미지 데이터를 바이너리 형태로 저장했을 때 수행시간이다. 우선 두 경우 모두 모든 어플리케이션에서 클라이언트대비 수행 시간이 급격히 낮아진 것을 확인 할 수 있다. 각각의 수행시간에서 서버 수행시간을 제외하면 연산 오프로딩의 오버헤드를 측정 할 수 있다. 캔버스를 문자열 형태와 바이너리 형태로 저장하였을 시 수행시간을 비교해보면 문자열로 저장하였을 때 보다 바이너리 형태로 저장한 경우가 수행시간이 더 적게 걸린 것을 확인 할 수 있다. 그 이유는 캔버스 이미지 데이터의 크기가 문자열에 비해 줄어들어 전체 전송시간이 짧아져서 오프로딩 성능이 향상했다.

캔버스 저장 방법에 따른 효과를 조금 더 자세하게 살펴보기 위해 GoogLeNet 어플리케이션을 대상으로 캔버스 저장 방법에 대해서 추론 연산시간을 제외한 오프로딩 수행시간을 측정했다. 그림 7은 앞서 말한 캔버스 저장 방법에 따른 오프로딩 수행시간을 비교한 그래프이다. 클라

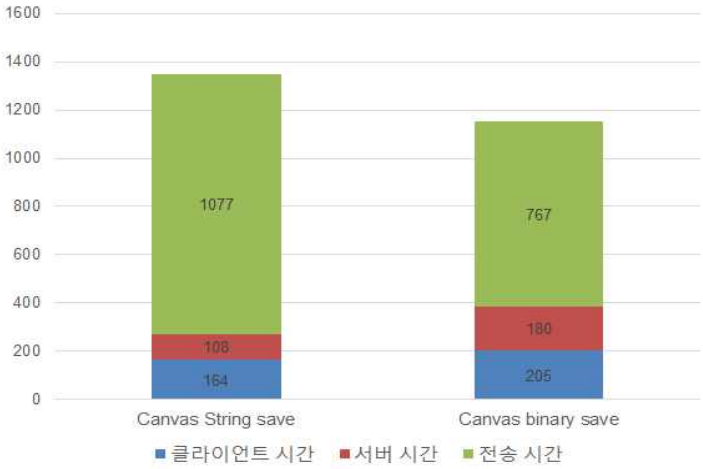

그림 7 오프로딩 수행시간 측정(연산시간 제외)

이언트 시간은 클라이언트에서 스냅샷을 생성 및 복구하는데 걸리는 시간이며 서버 시간은 서버에서 스냅샷을 생성 및 복구하는데 걸리는 시간이다. 전송시간은 스냅샷과 캔버스 이미지를 전송하는데 걸리는 시간이다. 두 그래프를 비교해보면 캔버스를 바이너리 형태로 저장했을 시 이미지 데이터를 디코딩하는 시간이 포함되어서 클라이언트와 서버에서의 시간이 약간 증가하는 것을 볼 수 있다. 또한 전송 데이터의 크기가 줄어들어 총 전송시간이 감소하는 것도 볼 수 있다. 결과적으로 증가한 클라이언트, 서버 시간보다 감소한 전송시간이 더 커서 전체적으로 캔버스 데이터를 바이너리 형태로 저장하는 방법이 오프로딩 수행시간을 감소시키는 것을 확인 할 수 있다.

제 3 절 모델 전송 방법에 따른 측정 결과

그림 6은 모델 전송 방법에 따른 추론 수행시간이다. 이 때 캔버스 저장 방법은 바이너리 형태로 저장해서 전송하는 방식으로 측정하였다. Model send by offloading은 모델을 오프로딩 시 전송하는 방식이며,

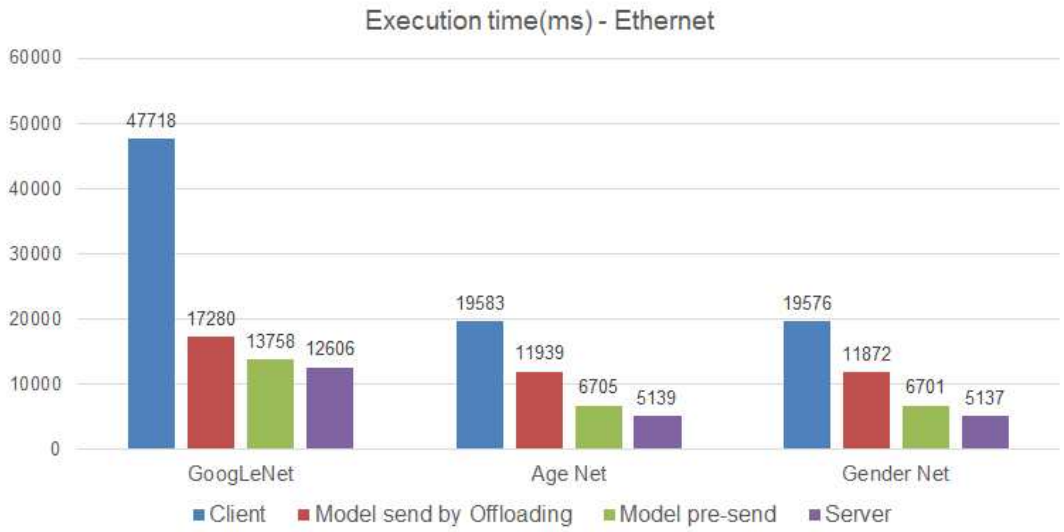

그림 8 모델 전송 방법에 따른 추론 수행시간

model pre-send는 Web worker를 이용하여 어플리케이션 시작 시 모델을 전송하는 방식이다. 이 때 측정 시점은 모델이 서버에 전송이 된 뒤 오프로딩을 사용할 수 있을 때 측정한 것이다. 우선 두 경우 모두 모든 어플리케이션에서 클라이언트 대비 수행 시간이 급격히 낮아진 것을 확인 할 수 있다. 모델 전송 방법에 따른 추론 수행시간을 비교해보면 모델을 오프로딩 시 보내는 것 보다 미리 보내는 것이 수행 시간이 더 적게 걸리는 것을 확인 할 수 있다. 그 이유는 모델 전송시간이 미리 보내는 경우 수행시간에서 제외가 되어서이다. 실제로 모델 전송시간만 측정해본 결과 두 전송 방법의 수행시간 차이만큼 걸리는 것을 확인 할 수 있다.

제 6 장 결론

스냅샷 기반 연산 오프로딩 기법은 스냅샷을 이용하여 어플리케이션의 상태를 쉽게 전송할 수 있어 프로그래밍이 쉽고 효율적이다. 하지만 기존의 스냅샷이 HTML5 캔버스 정보를 저장하지 못하는 문제가 있어서 캔버스를 사용하는 웹 어플리케이션에는 스냅샷 기반 연산 오프로딩을 적용할 수 없는 문제가 있었다. 본 논문에서는 기존의 스냅샷이 저장하지 못했던 캔버스 요소를 저장하기 위해 캔버스 정보를 복구하는 코드 생성 기술을 제안하여 캔버스를 활용하는 웹 어플리케이션에서도 스냅샷 기반 연산 오프로딩을 적용할 수 있게 하였다.

또한 머신러닝 어플리케이션을 대상으로 기존의 스냅샷 기반 연산 오프로딩을 적용할 때 스냅샷에 모델의 정보가 문자열 형태로 포함되어 오프로딩의 성능이 급격히 저하되는 문제가 발생한다. 본 논문에서는 이를 해결하고자 프레임워크에 함수를 추가하여 사용자에게 모델의 객체와 경로, 그리고 모델을 불러오는 함수를 입력 받았다. 이 정보들을 이용해 스냅샷 생성 시 스냅샷에 포함되던 모델의 정보를 포함하지 않고 바이너리 형태로 별도로 전송하여 오프로딩의 성능을 향상시켰다.

참 고 문 헌

[1] JinSeok Oh, Jin-woo Kwon, Hyukwoo Park and Soo-Mook Moon, “Migration of Web Applications with Seamless Execution”, Proc. of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 173-185, 2015

[2] Hyuk-Jin Jeong and Soo-Mook Moon, “Offloading of Web Application Computations: A Snapshot-Based Approach”, Proc. of the IEEE 13th International Conference on Embedded and Ubiquitous Computing, pp. 90-97 2015.

[3] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis and Mayur Naik, “CloneCloud : Bossting Mobile Device Applications Through Cloud Clone Execution”, arXiv preprint arXiv:1009.3088, 2010

[4] Mark S. Gordon, David Ke Hone, Peter M Chen, Jason Flinn, Scott Mahlke and Zhuoqing Morley Mao, “Accelerating Mobile Applications through Flip-Flop Replication”, Proc. of the 13th Annual Interanational Conference on Mobile Systems, Applications, and Service, pp. 137-150, 2015.

[5] Ian Hickson. HTML5 [online]. Available: https://www.w3.org/TR/2014/REC-html5-20141028

[6] Josefsson Simon, “The Base16, Base32, and Base64 Data Encodings,“ Internet Engineering Task Force, 2006.

[7] Masinter Larry, “The data URL scheme,” Internet Engineering Task Force, 1998

[8] Odroid, http://www.hardkernel.com/main/main.php

[9] Christian Szegedy et al., Going Depper with Convolutions, Proc. of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, 2015

[10] Gil Levi and Tal Hanssner, “Age and Gender Classsification

using Convolutional Neural Networks", Proc. of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, 2015

[11] CaffeJS, https://github.com/chaosmail/caffejs

Abstract

Snapshot-Based Offloading for Web Application Using Machine Learning Model

InChang Jeong

Dept. of Electrical and Computer Engineering

The Graduate School

Seoul National University

We propose a new approach to running machine learning web application on resource-constrained embedded devices, using snapshot-based computation offloading to general servers. Computation offloading is a method that dynamically migrates heavy computations to a server, letting the server perform complex computations. Snpashot allows saving the app execution state including the screen display in the form of another web app. We showed that the proposed offloading works for real web apps, with a performance comparable to running the app on the client and the server.

keywords : Computation offloading, Snapshot, Machine learning, Web application, Canvas

Student Number **: 2015-22809**